CS 5830 Homework 4

Instructor: Thomas Ristenpart
TAs: Andres Fabrega, Sanketh Menda, and Dhrumil Patel
Due: 16 November 2022

Submission and Grading

Submit your written answers to Gradescope and your code answers to CMSX. You must submit appropriate answers to the both submission sites for full credit.

Problems in this assignment are of two types:

1. *Attacking cryptography.* Given an incomplete template file, you need to fill in the missing pieces to mount an attack, e.g. retrieving a flag or forging a ciphertext.
2. *Talking cryptography.* Talk us through how you attacked a given cryptographic primitive or how you would hypothetically attack it.

The exact materials and the format you would need to submit them in for a given problem are indicated at the end of the problem.

Late Days

Homeworks are due on the due date by 11:59pm EST. You can use in total 3 late days throughout the semester.

Academic Integrity

Feel free to refer to external materials, but as usual please acknowledge them in your writeup in accordance with the Cornell Code of Academic Integrity.

Server Availability

To query the server, you'll use the same username and API key throughout this semester. Your username is your Cornell NetID (like `sm2289`) and you can find your assigned API key on CMSX, in the *Grading Comments & Requests* section of the placeholder assignment called `API_keys`.

We made a best effort to ensure the reliability of the challenge servers, but unfortunately, due to the nature of software and the internet, there may be disruptions. If you run into a disruption (the server is not responding, it is providing an error code, etc.), please do not panic, and reach out to the TAs (`@sanketh`, `@Andres`, or `@Dhrumil Patel`) on Slack. If this happens within 24 hours of the deadline, we will extend the deadline to accommodate for the inconvenience.

Verbosity for Written Answers

Generally, we prefer concise answers.

For a question like "Explain how you mount the attack?" we are expecting a high-level, but precise description of the attack in 3-10 sentences. Here is an example of a good answer.

> I started with an initial guess of the 16-byte all-zeros message. Then for each of the 16 bytes, I did X. Then did Y. Since the success of this is not guranteed, I repeated the process 20 times and aggregated. With the confidence, I did Z to recover the `FLAG`.

For a question like "Estimate the number of queries your attack issues?" we want you to think about how many queries you *really* need, and expect a high-level answer with ballpark numbers. Here is an example of a good answer.

> My attack does 5 queries initially for [purpose]. Then it does 1 queries per `FLAG` byte. In sum, for a flag of length N, it needs 1 + 5/N queries per `FLAG` byte.

Flag Notes

Each problem includes a description of the expected flag. For instance, it may say that it has 10 alphanumeric characters. Feel free to use that to restrict your search space, and to verify that you found the right flag (if you find something that meets the flag description, it is overwhelmingly likely that you found the right flag).

During grading, we will run your code against a different flag that nevertheless meets the description. So please make sure that you are not hardcoding assumptions regarding your specific flag.

Question 1. ECDSA Nonce Reuse

RSA is based on integers and integers have *a lot of structure* allowing fast attacks. Concretely, the time to factor an n-bit integer is not 2^n but something more like 2^{n^c} for some constant $c < 1$,[1] and in practice, this means that for 128-bit secure signatures we need 3072-bit RSA. In contrast, elliptic curve groups have almost no structure, they are just *groups*. This lack of structure allows us to build schemes that essentially require brute-force attacks, to "break" an n-bit curve, you need about $2^{n/2}$ time. Indeed, with ECDSA, we can get 128 bits of security with 256 bit private keys. So, it should come as no surprise that ECDSA is well standardized and widely used in practice.

Before reading forward, please skim the ECDSA Wikipedia page to familiarize yourselves with the scheme and the notation.

When using ECDSA, the first thing to keep in mind is that the nonce must be kept secret, otherwise one can easily recover the secret key from it. Denote by sk the secret key and $pk = g^{sk}$ the public key. Given a signature (r, s) for message m, if we know the nonce k, we can recover the secret key in the following way:

$$s = k^{-1} \times (H(m) + sk \times r)$$
$$sk = (s \times k - H(m)) \times r^{-1}$$

Even though, not properly selecting the nonce can also be catastrophic. Examples include the Sony PS3 hack, Bitcoin weak signature, and even a pretty recent one, the 2021 Anyswap Exploit. All these hacks result from a bad implementation: *nonce reuse*. If two signatures are generated using the same nonce, the secrect key can be recovered. Give two signatures (r, s_1) and (r, s_2) of messages m_1 and m_2 respectively using the same nonce k, we can recover k in the following way:

Since $s_1 = k^{-1} \times (H(m_1) + sk \times r)$ and $s_2 = k^{-1} \times (H(m_2) + sk \times r)$,
Substracting both sides, we have $s_1 - s_2 = k^{-1} \times (H(m_1) - H(m_2))$,
Therefore $k = (s_1 - s_2)^{-1} \times (H(m_1) - H(m_2))$.

In this problem, we will reproduce this attack. Let's pick the popular SECP256r1 elliptic curve, also known as NIST P-256 or PRIME256v1 by different organizations, and choose SHA-256 as our hash function. The vulnerable EDCSA implementation reuses a particular nonce with a significant probability.

Problem Oracle. We are given two oracles,

1. `getSignature` takes nothing and returns a random message and its corresponding ECDSA signature, and with high probability it reuses a nonce.
2. `verifySignature` takes a ECDSA signature of message `Hello from {NetID}`, and returns whether the signature is valid.

The Goal. We want to generate a valid ECDSA signature for message `Hello from {NetID}` by recovering the ECDSA secret key in this flawed implementation.

Query Estimates. This attack should ideally take a few queries, across all the oracles.

[1] See Carl Pomerance's A Tale of Two Sieves for a nice overview of these ideas.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw4_q1_ecdsa_nonce_reuse.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit it to CMSX.
- (5 pts) *Written answer.* Submit answers to the following questions to Gradescope:

  1. (3 pts) Explain how you mount the attack?
  2. (2 pts) Estimate the number of queries your attack issues?

Question 2. Attacking Textbook RSA

Introduction. Blake runs IT for a tax consultancy. Before the pandemic, the tax consultants would meet with clients in person, and if a client wanted to share digital documents with the tax consultants, they would hand them over in a USB drive. Then the pandemic happened. Overnight Blake had to build a system for clients to securely share their tax documents with the consultants.

First, Blake recognized that they need asymmetric cryptography, because clients may not have already established secure keys with the tax consultants. Second, Blake recognized that they need symmetric cryptography to actually send the documents, asymmetric cryptography is great for establishing a key but absurdly slow for actual communication.

After some Googling Blake chose AES128-GCM as the symmetric algorithm. Blake also chose to use the battle-hardened implementation from the Golang standard library. For the asymmetric algorithm though, Blake wanted to use RSA, and they knew from an undergraduate abstract math class that RSA was just exponentiation so wanted to implement it in-house.

So, the resulting system looks like following:

1. The client generates a random 128-bit AES128-GCM *session key* k.
2. The client encrypts their tax documents using AES128-GCM with this session key k to get the ciphertext c_D.
3. The client then encrypts the session key with the RSA public key (n, e) of the tax consultant by simple exponentiation $c_k := k^e \pmod{n}$.
4. The client then sends (c_D, c_k) to the tax consultant.
5. The tax consultant, first decrypts the session key using their private key (n, d) by simple exponentiation $k := c_k^d \pmod{n}$.
6. The tax consultant then decrypts the documents using k.

Blake deployed this system, and then immediately a bunch of consultants asked for a feature where clients, after uploading the documents, can check whether the documents were correctly received. So, Blake added another endpoint where a client could upload (c_D, c_k), and it returns whether the ciphertexts are valid.

You are in a tricky tax situation, you traded some crypto, and now you need tax help, so you go to this tax consultancy. You talk to them, think it is a good fit, and you send them your tax documents. Then they send you to this "confirmation" page where you can see that you can put in any ciphertexts, and it'll tell you whether it is valid. Snooping around, you also realize that they are using textbook RSA. You recall running into a paper years ago showing this configuration is vulnerable. You dig up the paper and indeed, some researchers showed that a similar configuration was vulnerable. Furthermore, you check, and this tax consultancy has a bug bounty, so now you want to develop a proof-of-concept, and maybe it'll even cover the tax preparation costs!

Problem Oracle. We are given two oracles,

1. `getWireData` takes nothing and returns an RSA public key (n, e), an encryption of `FLAG` c_D, and an encryption of the session key c_k. For simplicity, the oracle uses a fixed all-0s 12-byte nonce for AES128-GCM, which is ok because a new session key is generated for each query.

2. `checkConfirmation` takes an encryption of `FLAG` c_D and an encryption of the session key c_k, and returns whether it is valid.

For your reference, we provide a python implementation of the oracles in `qqrsa.py`. Feel free to utilize it for debugging.

The Goal. The goal is to recover the `FLAG`. See Section 4.1 of this paper for a description of how the researchers attacked the similar system.

Query Estimates. This attack should ideally take a few hundred queries, across all the oracles. And, is expected to take a few minutes of CPU time.

Flag Format. `RSA{[2 random bytes in hex]}`, like `RSA{ffff}`.

What to Submit.

- (2 pts) *Source code.* Copy the file `hw4_q2_rsa_cca2.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit it to CMSX.
- (3 pts) *Captured flag.* Put the captured `FLAG`, just the `FLAG` no quotes or special characters in text (not hex) in a new text file `hw4_flag_rsa_cca2.txt` and submit it to CMSX.
- (5 pts) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (3 pts) Explain how you mount the attack?
    2. (2 pts) Estimate the number of queries your attack issues?
- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (1 pts) Describe, at a high-level, how to implement a secure hybrid encryption scheme?

Question 3. Bleichenbacher's RSA e=3 Signature Forgery

In the previous attack we saw how textbook RSA is broken. The current best practice is to avoid RSA altogether as much as possible since it is so hard to implement correctly.[2] But, RSA is still widely used, with current *certificate transparency* data showing that over 90% of all certificates on the internet use RSA signatures.

Recall that Textbook RSA signatures with a modulus N, you take a message m represented as an integer, raise it to the decryption/signing exponent d, and the result $\sigma := m^d \pmod N$ is the signature. The verifier raises the signature σ to the encryption/verification exponent e to get back $m = \sigma^e \pmod N$ and verify that it matches the given message. First, observe that we don't have to have e and d be variables, we can fix one of them to be some constant. Of course, we shouldn't fix d, so let's fix e. $e = 0$ and $e = 1$ are bad choices. $e = 2$ seems okay, but it also doesn't work (why?). But, $e = 3$ seems to work and indeed it was a very popular choice. Second, notice that this only works when $m < N$. In practice N is a few thousand bits (usually 1024, 2048, or 4096), so we cannot sign messages longer than a few hundred bytes. Thus, we take a *hash-then-sign* approach where we hash the message and then sign the hash, and since hash outputs are small (usually 256 – 512 bits) we can sign arbitrary length messages.

But, what if you wanted to add some context information (like the hash used), so back in the day, people developed a "padding" scheme called *PKCS #1* (sometimes called *PKCS #1 v1.5*) where you'd encode the hash of the message with the context information like the hash used in a telcom-era format called *DER*.[3] The resulting encoding looks like

```
0x00 || 0x01 || 0xff ... 0xff || 0x00 || HashAlgorithmId || Hash(m)
```

Where we put enough `0xff`s till the whole message has the same length as the modulus N.

Again, why do I keep mentioning all these seemingly irrelevant details, why care about the `0xff`s. Well, once again, that tiny implementation detail being wrongly implemented leads to an attack.

A lot of implementations, most recently python-rsa, didn't check that the encoded message has the correct number of `0xff`s. What this means is that you can construct a message that looks like

```
0x00 || 0x01 || 0xff ... 0xff || 0x00 || HashAlgorithmId || Hash(m) || JUNK
```

without enough `0xff`s, and it would verify as correct padding. This combined with the choice $e = 3$ can be disastrous where you fix `JUNK` so that the resulting message m_f is a perfect cube and produce the forged signature $\sqrt[3]{m_f}$.

In this problem, we use SHA-256 and its `HashAlgorithmId` is

```
Sha256AlgorithmId = "\x30\x31\x30\x0d\x06\x09\x60\x86\x48\x01\x65\x03\x04\x02\
    x01\x05\x00\x04\x20"
```

Problem Oracle. We are given two oracles,

[2] See, for instance, Ben Perez's Seriously, stop using RSA.

[3] If you are curious about the format, see Let's Encrypt's A Warm Welcome to ASN.1 and DER.

1. `getPkAndMsg` takes nothing and returns a modulus N of a 2048-bit RSA public key (recall that we fix $e = 3$), and a message `m`.
2. `verifyRSA` takes an RSA signature σ, and returns whether it's a valid signature of `m`.

The Goal. We want to generate a valid RSA signature for message `m` without knowing the RSA secret key.

Query Estimates. This attack should ideally take a few queries, across all the oracles.

What to Submit.

- (5 pts) *Source code.* Copy the file `hw4_q3_rsa_e3.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit it to CMSX.
- (5 pts) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (3 pts) Explain how you mount the attack?
    2. (2 pts) Estimate the number of queries your attack issues?
- (1 pts extra credit) *Written answer.* Submit answers to the following questions to Gradescope:
    1. (1 pts) If N is of 1024 bits, can your attack always succeed? Why?

Evaluation

We would love it if you could provide us feedback to improve future homeworks. Please provide answers to the following questions on Gradescrope.

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours did you spent on this homework?
- Did you feel that the coding was too much, appropriate, or not enough?

And feel free to include additional feedback (on the homework or logistics in general). Also, this is not a tight deadline, we'd be delighted to hear your feedback via Slack before or after the deadline.